U.S. Department of Commerce
National Oceanic and Atmospheric Administration
National Weather Service
National Centers for Environmental Prediction
5830 University Research Court
College Park, MD 20740-3818

Office Note 503
https://doi.org/10.25923/3gp4-wj51

A DOCUMENTATION OF THE NMMB's NESTING CAPABILITIES AND MECHANISMS

Thomas Black

Environmental Modeling Center

May 5, 2020

email: tom.black@noaa.gov

ABSTRACT

A comprehensive nesting capability has been built into the NMMB. This note provides highlights of the key aspects of this capability. A global or regional parent domain can contain multiple static and/or moving nests that are 1-way or 2-way (upscale) interactive with their parent. Static nests can telescope through as many 'generations' as desired. Each moving nest can contain one moving nest within it but the innermost moving nest cannot telescope further. Each of these aspects of the nesting is described. The NMMB's nests are currently used in operations in the NAM and in HMON. The purpose of this note is to describe the fundamentals of how the NMMB's nesting works and basic aspects of how it is built therefore no forecast results are included.

1. Nest fundamentals

a. *The grids*

All nests are parent-oriented, i.e., they lie on the same grid projection of the Arakawa B-grid as their parents. For static nests it would be possible in principle to use free-standing nests that lie on their own projections that are independent of their parent. Generation of interpolation weights between parent and child would be needed only once during the initialization step. However such free-standing nests using two-way interaction would be impractical for moving nests since the cost of generating new interpolation weights for all points on the nest with respect to the parent would have to be generated every time a nest moved and that would be prohibitively expensive.

Figure 1 depicts a nested domain lying on its parent's domain. The x and y directions are the same in both domains indicating the projections are identical. Also shown is how the southwest corner mass point in the nest always lies on a mass point of the parent. This remains true for moving nests in that whenever they shift position their southwest corner mass point moves from one parent mass point to another. The vertical dimension is not considered here other than to note that the parent and all of its children must have identical vertical structures.

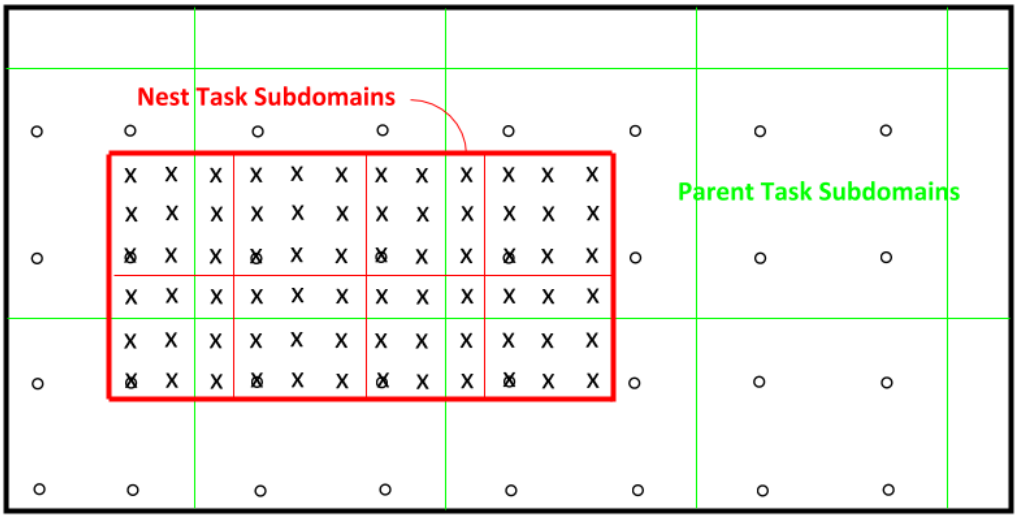

Figure 1. A nest domain (dark red) with its task subdomains (light red) lies on a portion of a parent domain (black) with its task subdomains (green). Mass points on the nest and parent domains are denoted as X's and O's, respectively.

The number of nest grid increments overlain by one parent grid increment can be either odd or even. The location of the parent's and the nest's mass and velocity points relative to each other is critical in properly describing the interaction of the two domains. Figure 2 shows these relationships for both odd and even ratios of grid increments.

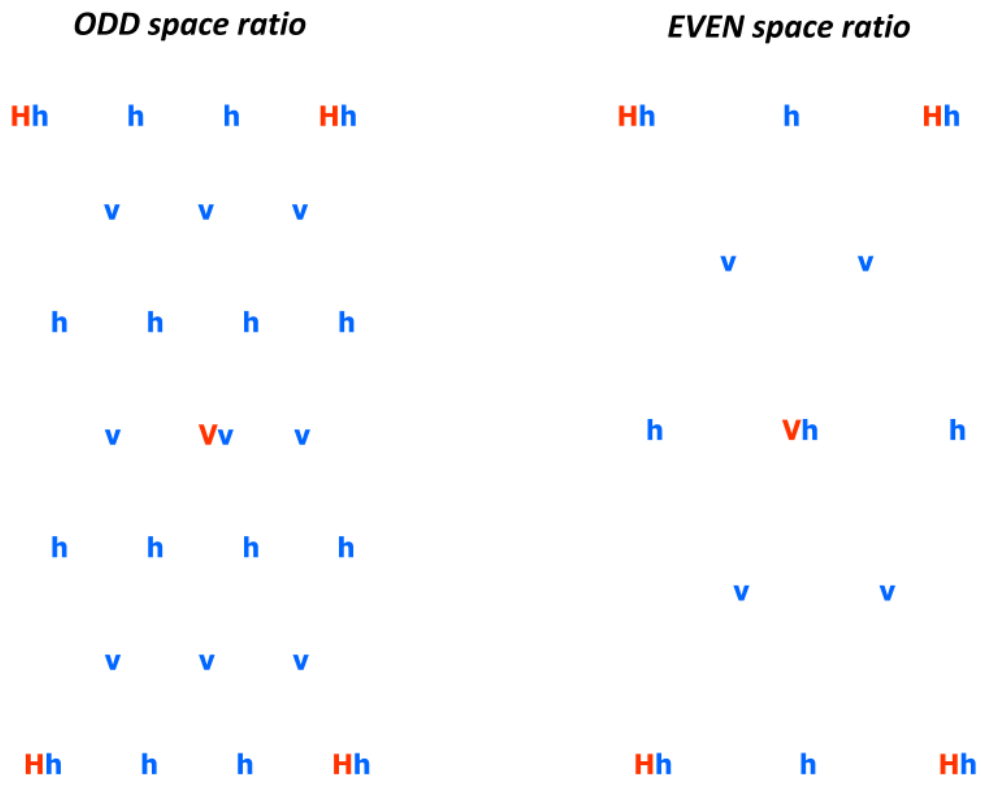

Figure 2. Mass and velocity points on parent (red) and nest (blue) grids. Mass and velocity points are H / h and V / v for parent / nest grids, respectively.

Note that for odd grid increment ratios parent mass points always coincide with nest mass points and parent velocity points always coincide with nest velocity points. For even ratios all parent mass and velocity points lie on nest mass points on the nest grid.

b. The two modes of integration

There are two fundamental ways that a parent and its children can interact. In 1-way mode the interaction consists solely of the parent's computing and sending boundary data to its children. That data is valid at the *end* of the parent's current timestep and the children receive it when they are at the *beginning* of that parent timestep. With that information the children can then interpolate boundary data linearly in time between the start and end of that given parent timestep. The parent computes this data upon reaching the end of each of its dynamic timesteps, sends it to the children with non-blocking sends, then immediately proceeds with its own integration. When the children reach the end of a parent timestep the data received from the

parent valid at the end of that parent timestep becomes the data valid for the beginning of the following parent timestep. Figure 3 shows a schematic of this process.

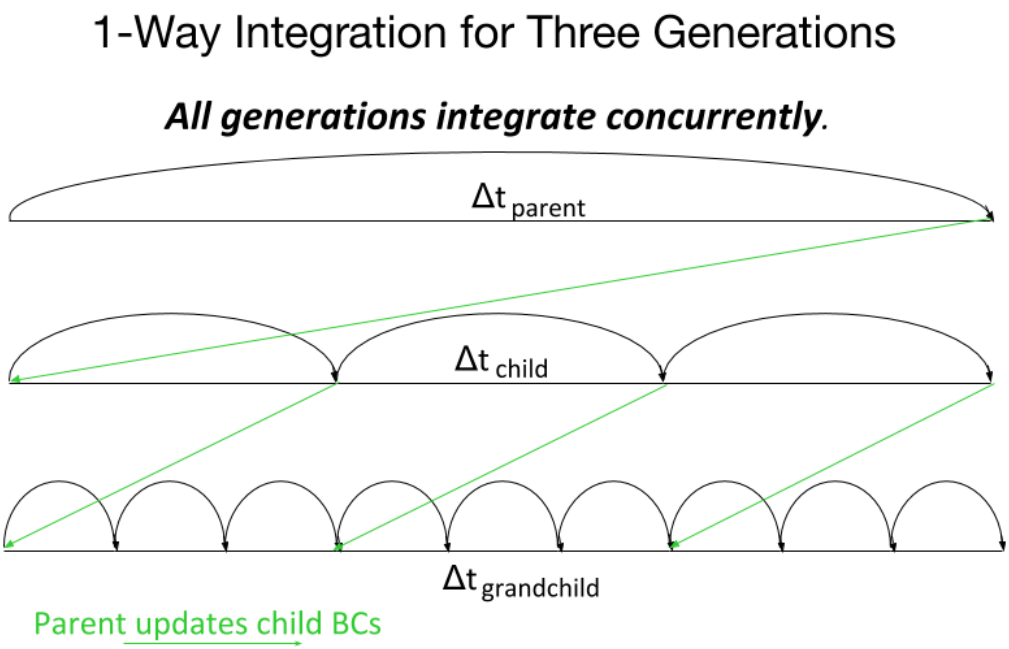

Figure 3. Parents send boundary data from the end of each of their timesteps to their children. Time is increasing from left to right.

In 2-way mode the parent still sends boundary data in the same manner but in addition the children send their own data from the interior of their domains to their parent. In theory this enhances the skill of the parent's forecast since the children's data is higher resolution although it is averaged on the nest grid before sending. After the parent sends boundary data to its children then it must wait until its children reach the end of that same parent timestep and send the parent their interior data and only then can the parent proceed. This dictates that domains on only one 'generation' at a time can execute through a parent timestep. Some models circumvent this waiting by making extrapolations of the inter-generational updates but such approximations are considered to be potentially too deleterious to the forecast and are rejected in the NMMB. This more complex process is shown in Figure 4.

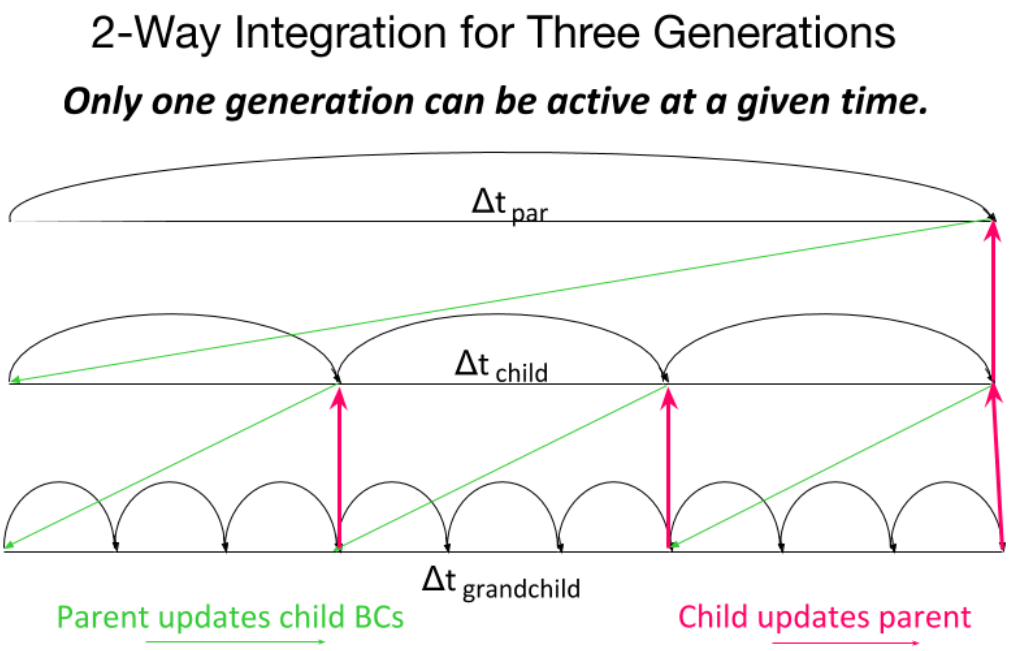

Figure 4. Parents send boundary data from the end of each of their timesteps to their children while the children send their interior data to their parents also at the end of each parent timestep.

c. Task Assignments

Like many models the NMMB segregates its MPI tasks into those used for the forecast computation and those used for asynchronous I/O. The latter may be referred to as quilt tasks in that they piece together the memory-distributed output into fields spanning the entire domain. There are two fundamental ways in which the NMM-B's MPI compute tasks are used. The first and simplest is called 'unique' which is used for 1-way interaction. In this case defined sets of tasks are assigned to each of the domains and each task will only ever operate on the one subdomain with which it is originally associated. An example of task usage in a set of 1-way nests is depicted in Figure 5 where 72 compute tasks are available.

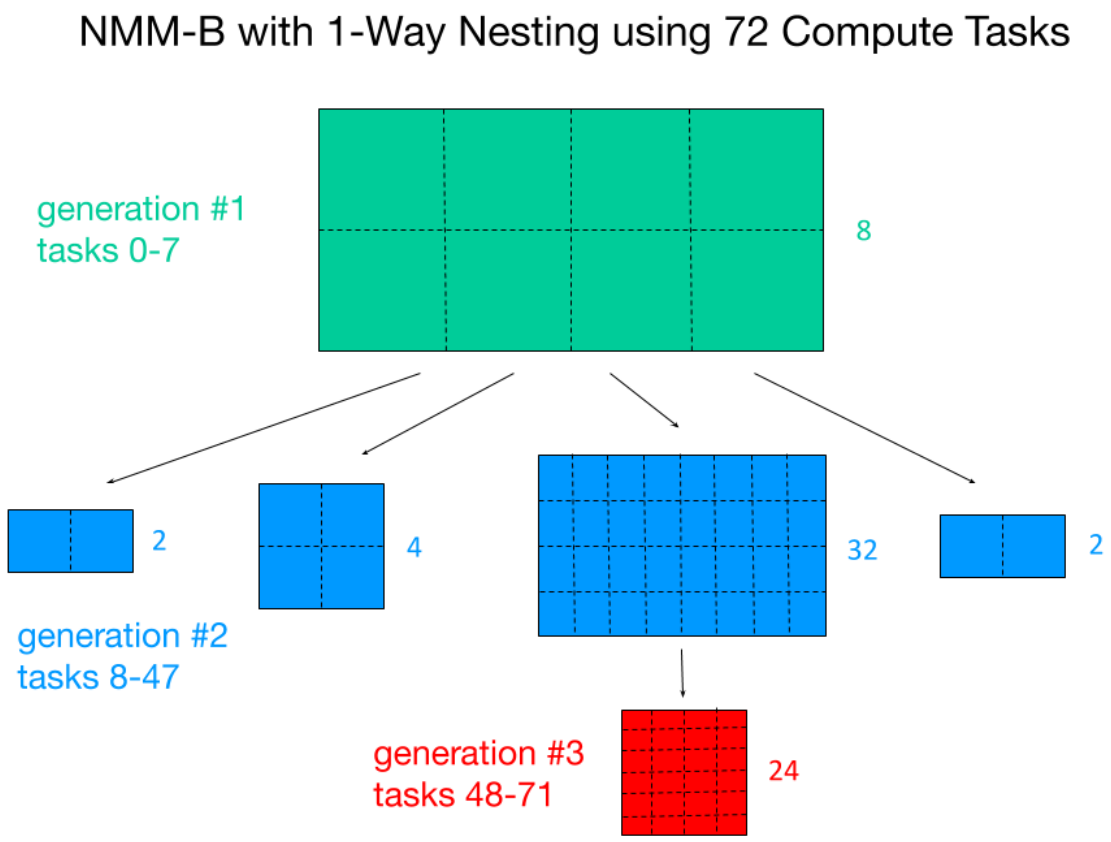

Figure 5. All available compute tasks are uniquely assigned to the various domains.

For each specific application a first approximation of the task count for each domain is made by taking the total number of points in the horizontal on each domain then multiplying by the ratio of the uppermost parent's timestep to the timestep of the given nest. Sum those values for all domains to get a total amount of work being done then determine each domain's fraction of that total. Multiply that fraction by the total number of available compute tasks to obtain the task count for each domain. For example assume the desired setup has an upper parent domain, a child nest, and a grandchild nest and there are 1000 available compute tasks. The upper parent has a timestep of 36 sec and has 10000 points in the horizontal. Its child has 8000 points and a timestep of 12 sec while the grandchild has 7000 points and a timestep of 4 sec. Then the products of point counts times timestep ratios is:

Upper parent: 10000 x (36 / 36) = 10000
Child: 8000 x (36 / 12) = 24000
Grandchild: 7000 x (36 / 4) = 63000

Grand total: 10000 + 24000 + 63000 = 97000

Upper parent fraction: 10000 / 97000 = 0.103
Child fraction: 24000 / 97000 = 0.247
Grandchild fraction: 63000 / 97000 = 0.650

Thus the first guess for the number of parent compute tasks is 1000 x 0.103 = 103 and similarly the counts for the child and grandchild are 247 and 650. To optimize the task counts the user needs to experiment. Parent domains will do some additional work by computing boundary values for their children every timestep which means task counts should be adjusted from the first approximation. The goal is to minimize wait time by both parents and children. Parents are one parent timestep ahead of their children when they send boundary data via non-blocking sends. This implies that parents should not run too fast or they could be ready to send new boundary data before the previous set of data has been received by the children thus requiring the parent to wait. On the other hand if a child runs too fast then it may be ready to receive new boundary data before its parent has sent it thus requiring the child to wait.

The second and more complex method of task usage is called 'generational' and it is used for 2-way interactive nesting where the nests send data from their domain interiors back to their parents. As explained earlier 2-way nesting means that parents send boundary data back in time to their children but the parent cannot then proceed as in 1-way nesting because it must wait to receive upscale feedback from its children. Therefore only one generation of domains at a time can execute its forecast through one of its parent's timesteps. Forecast speed is a top priority in operations therefore every effort must be made to minimize waiting by the compute tasks and forcing entire generations of domains to wait would be a very serious waste of resources. In the NMMB this problem is minimized by allowing processor cores that would otherwise be waiting to instead execute on domains within the currently active generation.

The strategy for generational task assignments begins with using ALL available compute tasks in the generation of domains that is deemed to be the most expensive computationally. Then in the remaining generations reuse as many of the compute tasks as needed to minimize that generation's runtime while bearing in mind that using too many tasks on a given domain can lead to a slowdown if the halo exchanges between the task subdomains take too much time. Therefore even though all tasks might not be used at all times the forecast still runs as fast as it possibly can. An example of 2-way task use on three generations of domains is seen in Figure 6 where again 72 compute tasks are available as in Fig. 5. For the sake of illustration it is assumed that in this forecast generation 2 was determined to be the most computationally expensive.

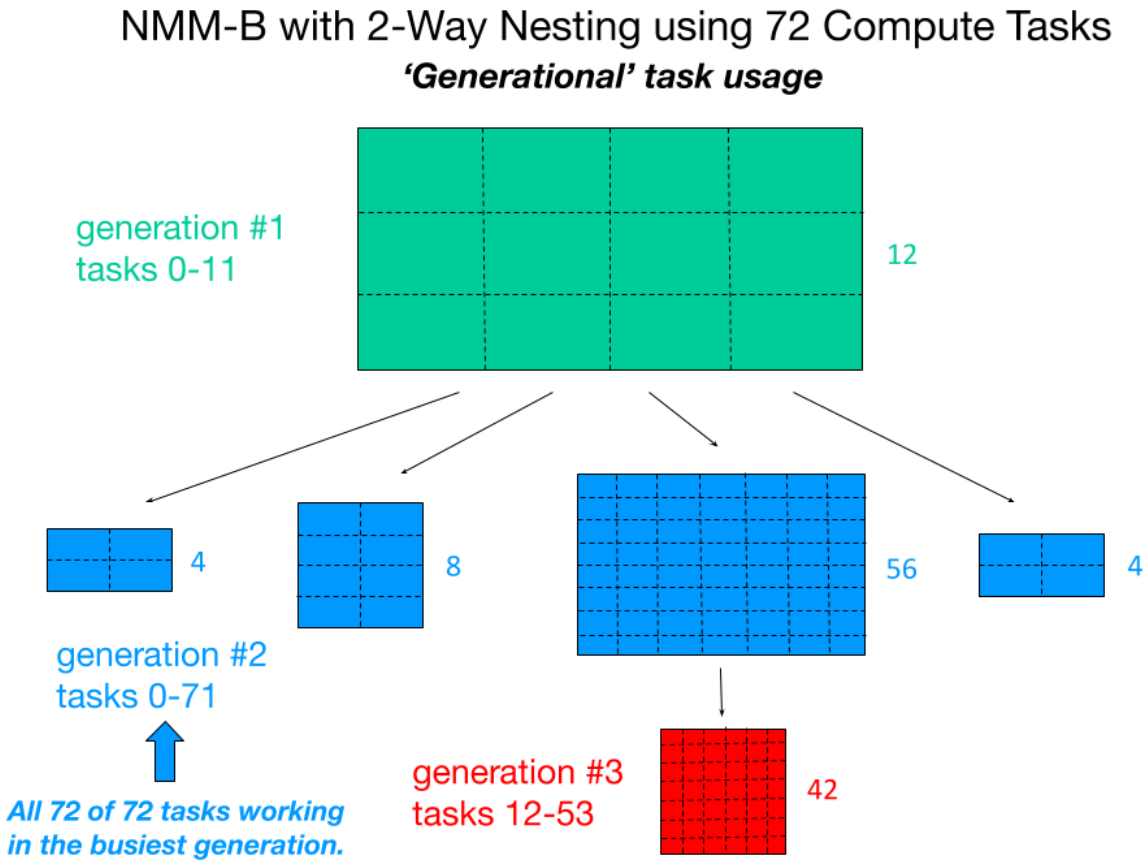

Figure 6. All available compute tasks are assigned to the 2nd generation since it is the most expensive computationally. Some of those tasks are then reused in the remaining domains in the other generations.

A task is not allowed to lie on more than one domain in a generation because all domains within each generation run concurrently. Also it is important to note that the quilt tasks must remain fully separate from the compute tasks or else the sending of the history/restart data from the compute tasks would not be asynchronous.

d. MPI communicators

All interaction between different domains is strictly between a parent and its children. Arrays of MPI communicators are created to handle those communications. ***Inter***communicators between a parent and each of its children are used for unique task assignments. They are the simplest type of communicator since the lead task on each domain has a rank of 0 and MPI sends/recvs use target and sender task ranks. For instance assume the global task ranks of a parent domain are 25, 26, and 27 while the global task ranks of one of its children are 52, 53, 54, and 55. Then the intercommunicator task ranks are 0, 1, and 2 for the parent and 0, 1, 2, and 3 for the child.

When using generational task assignments intercommunicators cannot be used since they require that each task exists in only one group, i.e., on each domain. Since generational task usage means tasks may lie on more than one domain then ***intra***communicators are required between parents and their children. To see how they work assume we have the following global task ranks on a parent and its child.

Parent - 3, 4, 5, 6
Child - 1, 2, 3, 4, 5, 6, 7

Now starting with the parent's tasks create a non-repeating union of all the tasks then translate that list into a monotonic one starting with 0.

Union - 3, 4, 5, 6, 1, 2, 7 - - > 0, 1, 2, 3, 4, 5, 6

Using that association then the two domains' ranks within the intracommunicator are as follows.

Parent - 0, 1, 2, 3
Child - 4, 5, 0, 1, 2, 3, 6

Since the parent's global ranks were listed first in the union they will always range from 0 to the total number of parent tasks minus 1 in the communicator. This makes communication from the child's perspective straight forward. However as seen in this example the child's task ranks in the union can be jumbled depending on how they overlie the parent tasks. Therefore the parent must determine and store the union ranks of each of its children in order to use the intracommunicators. Referring to the numbers above when a parent task wants to communicate with the child task with global rank 2 which is 6th in the union then the parent task would know that it must use the 6th value in the monotonized sequence which is 5.

e. The composite object

When running with generational task assignments in 2-way interaction some or all tasks will lie on more than one domain yet each task sees variables and array references in the source code such as T(i,j,k) (sensible temperature) and PARENT_SHIFT(n) (a parent's timestep in which it will shift plus the distance in i and j it will shift) that are naturally different for each domain a task is on. The question then arises as to how a task can differentiate between these variables and arrays on the multiple domains with which it is associated. If a task lies on three domains then it must have three different locations in memory for the T array. This problem is solved in the NMMB by using something called the composite object which is a derived datatype holding every variable that has unique values in the forecast. It allows any task on multiple domains to reference those variables generically as they appear in the code while pointing at the correct

memory locations of the variables for the particular domain. The following idealized code illustrates how this works. The type is defined along with generic versions of that type's components that are declared as pointers.

```
TYPE COMPOSITE
  REAL, DIMENSION ( : , : , : ) :: T
  INTEGER, DIMENSION ( 1 : 3 ) :: PARENT_SHIFT
END TYPE COMPOSITE

REAL, DIMENSION ( : , : , : ), POINTER :: T
INTEGER, DIMENSION ( : ), POINTER :: PARENT_SHIFT
```

In a setup routine a pointer of type COMPOSITE is declared and allocated to the total number of domains (NUM_DOMAINS).

```
SUBROUTINE SETUP

TYPE ( COMPOSITE ), DIMENSION ( : ), POINTER, SAVE :: CPL_COMPOSITE
ALLOCATE ( CPL_COMPOSITE ( 1: NUM_DOMAINS, stat = ISTAT ) )

END SUBROUTINE SETUP
```

Not shown here is the subsequent filling with actual data of all the components of CPL_COMPOSITE for each domain the given task is on. Then in a subroutine called POINT_TO_COMPOSITE a pointer of type COMPOSITE is declared and pointed at that part of the composite object holding the variables relevant to the task's current domain given by MY_DOMAIN_ID. Finally the declared generic variables are pointed at the appropriate location in the composite object for the current domain's data.

```
SUBROUTINE POINT_TO_COMPOSITE ( MY_DOMAIN_ID )

INTEGER, INTENT (IN) :: MY_DOMAIN_ID    !<- - The ID of the task's current domain
TYPE ( COMPOSITE ), POINTER :: CC
CC => CPL_COMPOSITE ( MY_DOMAIN_ID )

T => cc%T
PARENT_SHIFT => cc%PARENT_SHIFT

END SUBROUTINE POINT_TO_COMPOSITE
```

Now the task is prepared for use of the generic variable names T and PARENT_SHIFT which by themselves in the source code offer no indication of which domain the task is on. Routine POINT_TO_COMPOSITE simply needs to be called to align those generic variables with the current domain's appropriate data locations in memory.

```
SUBROUTINE XYZ

CALL POINT_TO_COMPOSITE ( MY_DOMAIN_ID )

CALL MPI_RECV ( PARENT_SHIFT, 3, MPI_INTEGER, ……

END SUBROUTINE XYZ
```

In this case the MPI_RECV will always use the correct memory location of PARENT_SHIFT for whichever domain the task is on at the given time.

f. General initialization

The NMM gridded component is the ESMF component lying at the highest level within the model. It handles work associated with initializing the run. This includes a very large number of specific details related to the parents and the nests and because they are documented in the initialization procedure source code those details will not be covered here.

However one particularly important aspect of the initialization step will be mentioned. As stated earlier all of the NMMB domains are functionally equivalent. Each domain is an element in DOMAIN_GRID_COMP which is simply an array of the ESMF gridded components representing each of the domains. There is nothing special about the nature of the uppermost parent domain other than it must be domain #1 since that fact is often used in logic through the nesting code. The use of nesting means that both parent and child domains exist, therefore the routine that initializes DOMAIN_GRID_COMP must be called recursively. The reason is that for the sake of certain tests the option is available for parent domains to generate initial data for their children and so children must not call the initialize routine before their parent has done so or else they will try to read in data that does not exist. First DOMAIN_GRID_COMP is initialized for its array element 1 (the uppermost parent). After a barrier to make child domains wait then the initialization is called recursively within a loop over all the first generation children. In each iteration of the recursion the next generation DOMAIN_GRID COMP is initialized while its children wait then a loop over its children initializes each of them.

g. Configure files

The uppermost parent domain and all nests are functionally equivalent and all must have a configure file associated with them. Each of those configure files must be located in the working directory where the forecast will be executed. The code reads in each configure file and loads it as an ESMF configure object for later access to the files' settings via standard ESMF calls. The name of each configure file is:

configure_file_NN

where NN is a two-digit integer between 01 and 99 (an arbitrary value of 99 was selected as the largest number of different domains that can coexist in a given run). The total number of such files sitting in the working directory is counted and compared to the intended total number of domains that the user specifies in the configure files. If those two values are not the same then the run aborts with a message. The user specifies the length of the forecast in the configure files. When the files are loaded a check is done to be certain that the forecast length in all files is the same. If it is not then the run aborts with a message.

2. Timestepping

The execution of the timestepping in the forecast integration is in subroutine NMM_RUN (the Run step of the NMM gridded component) and it differs fundamentally between 1-way and 2-way nesting. In 1-way nesting each task belongs to only one domain and all domains run concurrently from the start to the end of the forecast where the parent always leads its children by at least one timestep as described above. In 2-way nesting at least some tasks lie on multiple domains but never on more than one domain per generation which means a loop over the generations must exist over partial timestep loops allowing tasks to return after the timestep is finished so they can participate in another generation's timestep(s) before switching generations again. The number of generations (NUM_GENS) in the outer loop is an integer greater than 1 only for 2-way nesting. Here is a simple schematic of the forecast timestepping.

```
main_block: DO WHILE ( .NOT.ALL_FORECASTS_COMPLETE )

  generations_loop: DO N = 1, NUM_GENS

    domain: IF ( MY_DOMAIN_ID > 0 ) THEN  !<-- Domain ID is 0 for 2-way nesting if task
                                          !     is not in generation N.
.
.
      CALL NMM_INTEGRATE( ……. )        !<-- The integration timestepping routine.

      IF ( ESMF_ClockIsStopTime ( CLOCK_NMM ( MY_DOMAIN_ID ), rc=RC ) ) THEN
```

```
          GENERATION_FINISHED ( N ) = .TRUE.    !<-- Task's fcst in generation N has
                                                !     finished the entire integration.
        ENDIF

      ENDIF domain

      IF ( ALL ( GENERATION_FINISHED ,NUM_GENS ) ) THEN  !<-- All of this task's
                                                         !     domains are finished?
        ALL_FORECASTS_COMPLETE=.TRUE.
        EXIT generations_loop
      ENDIF

    ENDDO generations_loop

ENDDO main_block
```

Now summarize what is happening here. The WHILE block called ‘main_block’ will continue to iterate until the given task has completed the forecast on every domain it lies on as indicated by the value of the logical flag ALL_FORECASTS_COMPLETE. That flag is initialized to false. Within that WHILE block is a DO loop called generations_loop that iterates over all the generations. If a task lies on a domain within the current generation of the loop then the subroutine NMM_INTEGRATE is called and the task will enter the model integration. Immediately upon returning from NMM_INTEGRATE the Clock is checked and the value of GENERATION_FINISHED for the current generation of the domain that is executing is set to true if that domain has finished its forecast. If the given task has finished its forecast in all generations it is running on, i.e., if all elements of the logical array GENERATION_FINISHED are true, then ALL_FORECASTS_COMPLETE becomes true. When ALL_FORECASTS_COMPLETE becomes true then the given task drops out of ‘main_block’ and finalizes its execution.

The core timestepping loop named ‘timeloop_drv’ occurs inside of subroutine NMM_INTEGRATE. The limits of that loop are simply the first and last timesteps of the integration to execute on the current domain. For 1-way nesting all domains integrate straight through from the start to the end of the forecast. Two-way nesting is different due to the children's feedback. All generations except the lowermost will execute only one timestep at a time then return since their domains cannot proceed until they receive internal updates from their children (see Fig. 4). The domains in the lowermost generation have no children and can thus execute a full N timesteps at a time where N is the number of timesteps within a single timestep of their parents. Due to the nature of the generational use of task assignments in 2-way nesting some tasks will enter timeloop_drv but not be allowed to integrate because: (1) parent domains must first recv 2-way exchange data from all of their children at the end of each parent timestep;

(2) child domains at the end of their parents' timesteps must be informed by their parent that the parent did recv exchange data from all its children meaning the given child can proceed because its parent is free to integrate to the end of its next timestep and send back BC update data. If a domain is both a parent and a child then both conditions (1) and (2) must be true for the task on the given domain to integrate another timestep.

3. Computation of nest boundary conditions

The most fundamental aspect of nesting is that a parent domain continually computes and sends lateral boundary data to each of its children at the end of every parent timestep as described above. Now we consider how that boundary data is generated. When a parent reaches the end of its current timestep it checks to be sure all nest boundary data has been received by the nests one timestep ago. If so then it proceeds with computing the new surface pressure on the nest boundary points since this is a fundamental quantity needed for generating boundary values for the child. With static nests each parent task determines in the initialization step which nest tasks have any portion of their boundaries on the parent task and exactly which nest points those are. For moving nests the parent must recalculate which nest boundary points must be updated every time any of its children shift position. In the general case note that more than one parent task may be updating boundary segments on a given nest boundary task subdomain.

The parent with its own topography must produce data for a nest's boundaries that accounts for the nest's topography. For nest mass points the parent begins by doing a bilinear interpolation of its surface pressure and its surface geopotential to each nest boundary point location which allows it to compute its model layer interface geopotentials over each of those nest points. Knowing the surface geopotential at each nest boundary point the parent then does a vertical interpolation linearly in log P between its interface pressure and geopotentials to find the log of the nest surface pressure and thus the nest surface pressure itself after exponentiation. If the nest surface lies below the parent surface then a quadratic extrapolation from the parent's surface pressure is done to obtain the nest surface pressure. This process is done for two rows along the nest boundary which then permits the computation of surface pressure at nest boundary velocity points by doing a 4-point average of the surrounding mass point values. Care is taken in the indices of these arrays given the nature of the semi-staggered B grid.

Now the parent bilinearly interpolates its layer interface pressures to the nest boundary point locations and vertically averages them to get the mid-layer pressures. Likewise it takes each of its variables that must be provided on the nest boundary and bilinearly interpolates them to the nest point locations. Given the values of the nest boundary point surface pressures the parent computes the mid-layer pressures in each nest layer. The parent can then do a cubic spline interpolation using the boundary variables' values in the middle of its layers over the nest point locations to compute the values in the middle of the nest layers over those locations. If the mid-

layer pressure of the nest is greater than the mid-layer pressure of the parent's lowest layer then a linear extrapolation is done. However if the extrapolation is large then unphysical values could result so it is moderated by an additional factor between 0 and 1 based on a hyperbola. For small extrapolations the factor is near 1 but as the extrapolation gets larger and larger the factor asymptotes to 0 thereby keeping the magnitude of the final extrapolated value from growing too large. When a temperature inversion is present in the parent then the values of T in the lowest two layers are first modified so that the lowest layer is a mass-weighted average of the three lowest layers and the next-to-lowest layer is a mass-weighted average of the lowest two layers. This is done to prevent skewing of the final value in the nest when extrapolation must be done.

Updated variables on the nest boundary are not sent to the nest individually. Instead for each side of the nest domain each parent task allocates two derived datatype objects, one for mass points and one for velocity points, that hold all updated variables for all points on all nest tasks for which the given parent task is responsible. For each nest task to be updated an unallocated pointer is pointed to vertical columns in each update boundary variable in subroutine PARENT_UPDATE_CHILD_BNDRY. The unallocated pointer is what is sent into the boundary update routine so that when it is updated via a vertical cubic spline the actual primary object is automatically filled in its proper locations. After all variables are updated then the parent tasks do non-blocking sends of the data to each nest boundary task that is covered by the parent tasks. The target nest tasks receive the strings of data which are from more than one parent task where the nest boundary crosses the edge between two parent task subdomains. Upon receiving the data the nest task separates the data received from parent tasks and combines it into unified segments on the boundary for each variable. There may be overlap between data from two different parent tasks due to their subdomain halos and if so then the values from the parent task with the smaller rank is always selected in subroutine CHILD_RECVS_CHILD_DATA_LIMITS. This eliminates randomness of data arrival in the nest and ensures that forecasts are bit reproducible.

4. Nest motion

By far the greatest complexity of nesting is related to the motion of moving nests. The most fundamental aspect of a domain's shift is its final position relative to the location of its own and its parent tasks' subdomain locations prior to the shift. Figure 7 is a schematic of the situation.

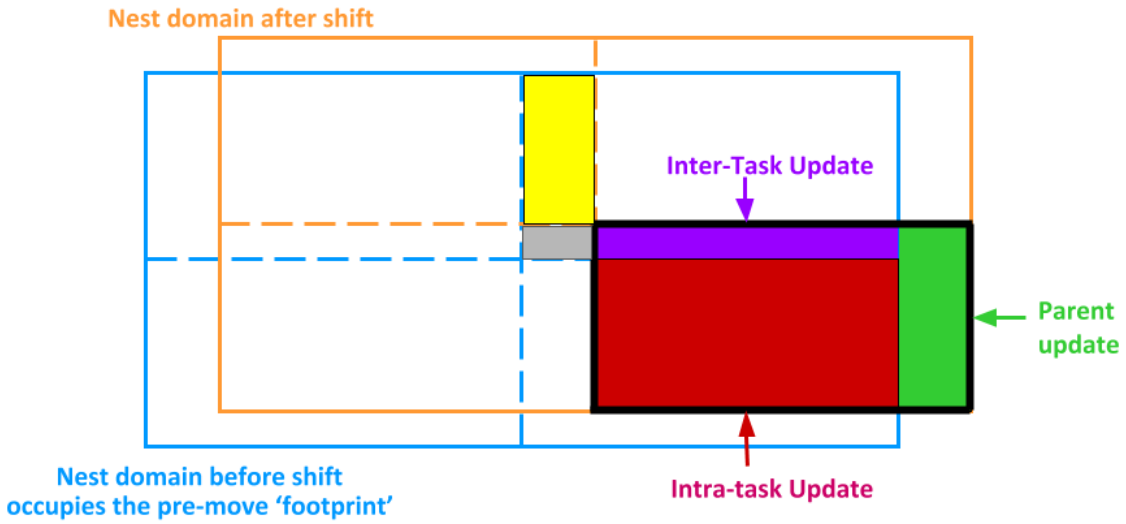

Figure 7. Three types of data motion indicated by red, purple, and green are required to update all of the points in the nest's southeast task after a shift to the northeast. The purple, gray, and yellow are regions the northeast task will use for intertask updates of the other three nest subdomains following the shift.

In this example the nest domain is covered by four MPI tasks and it has shifted to the northeast. In order to update the values in the nest's southeast subdomain there are three types of data motion that must take place. The area in red encompasses points that lie within the same subdomain before and after the shift so an intra-task update is needed there. The southeast task's points in the purple region have moved beyond that task's previous location therefore an inter-task update will be needed from the northeast task. Finally the points in the green region have moved completely outside of the nest domain's original position therefore they must be updated by one or more of the parent's tasks.

Begin by considering the intra- and inter-task updates that come from the nest's own tasks. These actions cannot be done in a simple sequence because if they were then data would be lost that was needed for one or the other type of update. Therefore they are done in the following order: (1) Data is gathered into ISend buffers for the inter-task shift and is sent to the target nest tasks; (2) The intra-task update is done; (3) The inter-task data is received and applied.

In preparing for the inter-task sends in step (1) all nest tasks must determine which of their points' data must be sent to which other nest tasks. Under normal circumstances the number of grid increments the nest shifts on its grid does not exceed a nest compute task's subdomain

dimensions. If that is the case and if the nest motion has both I and J components then each task except those on the trailing edge will send to three other nest tasks as seen by the purple, gray, and yellow areas in Fig. 7. All trailing edge tasks except the trailing corner will send to one task. The trailing corner task will send to none. If the motion has only an I or only a J component then each non-trailing edge task will send to only one task and the trailing edge tasks will send to none. However in the most general sense if the distance of the nest's motion exceeds the dimensions of its compute tasks and the halo points of the receivers are included in those points to be updated (to avoid doing repeated halo exchanges after the shift) then there are nine tasks that can potentially receive data from a given task that is sending.

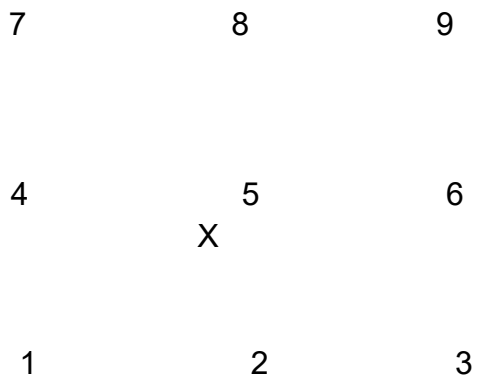

Figure 8. For a nest task at location X following a shift in position then there are nine nest tasks that can potentially receive inter-task update data from it.

In Fig. 8 the reason the outer tasks could receive data is that the halo region of receiving tasks is updated too while the sending tasks only send data from points in their compute regions so the target area of the tasks at the nine locations is slightly larger than the source area of the sending task at X. Additional details must be considered especially with respect to domain edge values at mass and velocity points and dynamical tendency values in the domain's 2nd outermost row. Those details will not be discussed here. A loop over all nine potential receiving tasks is executed to determine which of them in their post-shift positions overlap the area of the sending task X in its pre-shift location. For those that do overlap, the exact number of points within the overlap region is found and then the total number of real and integer words for all update fields can be computed. Next a loop over all update variables themselves is executed and the values at the overlap points for each receiving task are saved into buffers. When all update data has been saved then the real and the integer data are sent to the target receivers in non-blocking sends.

Step (2) in the update process following a shift is intra-task and is thus the simplest. This handles those points in the task subdomain whose post-shift locations remain within the pre-shift

footprint of the same task which naturally means there is no need for MPI. Again update points include the subdomain halos but the source data comes only from compute points. The updates are done simply by considering each relevant variable's array, looping through the affected points and setting the new value at each location to the current value at a different location, i.e., A(i,j) = A(i+i_shift, j+j_shift). Care must be taken though. If the shift was northward then the J loop can iterate positively. However if the shift was southward then the J loop must iterate negatively or else source data later in the loop would already have been changed to updated values earlier in the loop. If the shift has no northward or southward component then the same notion is applied for the I loop, i.e., it must iterate negatively for a westward shift.

Step (3) of the update process is essentially the reverse of step (1). A loop over the nine potential sending tasks is executed. Overlap regions are determined to find which tasks are actually sending updates, word counts for all real and integer update data are computed, and then that data from each of the sending tasks is received and incorporated into each of the updated variables.

The most complex update following a nest's shift is for those points that move beyond the pre-shift footprint of the nest and which therefore must be given new values by the parent. In Fig. 7 these points are those within the green region for the task subdomain in the nest's southeast corner. For technical reasons the parent will also update any points that not only have moved beyond the nest's pre-move footprint but also those that have moved onto the outer two rows of that pre-move footprint. This is related to the nature of the semi-staggered B grid and to the fact that some key dynamical tendencies are not computed on the 2nd row around the edge of the nest domain. While most of these update regions are rectangular, some nest tasks after the shift will lie on a corner of the pre-move footprint meaning that the update region the parent will handle is a rectangle with a wedge missing. These areas require complicated logic to determine which points on individual nest task subdomains are updated by which parent task subdomains. Nest task subdomains that shift to a corner will be updated by a minimum of one and a maximum of four parent tasks. Rather than have either the parent or the nest tasks do all the bookkeeping (which points on which nest task subdomains are updated by which parent tasks) and then share it via MPI with the other domain all update bookkeeping is done independently by the nest and the parent tasks. This approach avoids additional MPI communication and also allows each of the two domains' bookkeeping results to serve as a check on the other's since if they disagree then the sends and recvs of data would fail. After the parent finishes its bookkeeping then depending on the variable it generates the update data using either nearest neighbor values, plain bilinear interpolation, bilinear interpolation accounting for the presence of the sea mask, or bilinear interpolation accounting for the presence of the land mask. For 2-way interaction the number of nest tasks that will update a given parent task will change in time therefore ordinary arrays cannot be used efficiently to receive and store data from the child tasks that are sending. Instead each parent task interacting with a nest task creates a linked list for each moving nest with information from those nest tasks contained within each link. Each time any of the parent's

children shift then the linked list of those parent tasks associated with that child is deallocated and is built again based on the new positions of the two domains' task subdomains.

When a child nest has decided to move it will do so three parent timesteps after making the decision since the parent could be either one or two timesteps ahead of the child for 1-way interaction. The parent continually probes for messages from its moving children indicating when the child will move and containing the I and J components of the shift. The message is received by the parent at the end of a parent timestep while the child will have sent it from the beginning of an earlier parent timestep depending on the relative integration speeds of parent and child. The parent computes and sends the new internal child data for those child gridpoints that have moved over a new region of the parent grid as well as the new boundary data valid for their grids' new locations.

What has just been described is the shifting of a nest on its static parent's domain. Recall that at the beginning of this note it was stated that moving nests can telescope one more generation and can thus contain another moving nest. This setup is particularly important for tropical storms. The entire globe or a static basin over a large region of interest serves as an uppermost parent. On that static domain one or more moving nests follow their respective storms and should be large enough to cover much of their surrounding environment. Within each moving nest lies a smaller moving nest that encompasses the storm more closely to provide even more resolution. Allowing for moving nests within moving nests introduces another layer of considerable additional complexity. Only two fundamental aspects of moving-within-moving nests will be mentioned here. When a child nest wants to shift then it must inform its parent so that boundary data at the post-shift location can be prepared by the parent. When a parent (outer) nest has decided that it will shift it must notify the lead task of each of its moving children. This is required so that the children will be able to recompute parent-child task layout relationships which will change after the shift. It also forces the children to wait to receive the task update specifications for boundary data from the parent in its new position before they execute their normal receiving of boundary data from the future.

One particular problem of moving nests within moving nests is the possibility of collision between an inner nest and its outer nest's boundary. Naturally this must be avoided. When an inner nest determines it wants to shift a distance that will cause a collision then it instructs its parent that the parent must first shift far enough to avoid the collision. After the parent has completed the evasive shift then the inner nest can shift.

The source of storm motion within the model is the storm tracker which makes dynamical calculations during the forecast to determine how the storm moves. The storm tracker is called after an integer number of physics timesteps (= 20 sec) and in operations that number is 20 so the tracker is called every 400 sec. In order to avoid the possibility that an outer and inner nest will make different decisions about their motion given input from the storm tracker only inner

moving nests are directly tied to the tracker thus it is the inner nests that actually follow each storm. The outer nests always know the location of their inner nests so they simply follow the inner nests. The user specifies in the moving parent's configure file the greatest number of parent grid increments that its center can be from the inner nest's center. When that value is exceeded the outer nest will shift to bring the centers back together as closely as possible given certain constraints.

5. Fundamental integration sequence

Now that fundamental aspects of the nesting have been described the following primary steps in the execution of the NMMB forecast with moving nests using 2-way interaction with their parents are listed. If the nesting is 1-way interactive then those details related to 2-way can be ignored.

(1) Consider the beginning of a nest timestep coinciding with the start of a parent timestep. If there is 2-way nesting then check for the signals from parents and children to know if the execution can proceed into this timestep. Due to the nature of the generational use of task assignments in 2-way nesting some tasks will enter the main integration timeloop but not be allowed to integrate because: (i) Parent domains must first receive 2-way exchange data from all of their children at the end of each parent timestep; (ii) Child domains at the end of their parents' timesteps must be informed by their parent that the parent did receive exchange data from ALL of its children meaning the given child can proceed because its parent is free to integrate to the end of its next timestep and send back BC update data. If a domain is both a parent and a child then both of those conditions must be true for the domain to integrate another timestep.

(2) Immediately following (1) are these actions: (i) If a child has decided it wants to move then it sends a message to its parent informing it of that fact along with the location to which it is moving on the parent grid. That move must happen at a parent timestep in the future because the parent must provide data to some internal child points following a move and since the parent must always run ahead of its children (to provide the boundary data from the future) then the new data for those internal child points must also originate at a future timestep. That future parent timestep in which the nest will shift is also sent to the parent. (ii) If the current timestep is equal to the timestep in which the nest determined it wants to shift then the child now receives the parent data for its internal gridpoints that have moved over a new portion of the parent grid as well as the boundary data at the new location. (iii) All children receive boundary data updates from their parent from one parent timestep in the future.

(3) If 2-way interaction is turned on then the parent receives the upscale data sent from its children.

(4) The uppermost parent integrates one timestep and waits for 2-way upscale feedback. Then each of its children integrate one timestep and wait as well. Grandchild nests then integrate N3 timesteps where N3 is the number of their timesteps per timestep of their 2nd generation parent and send 2-way upscale data to the parent. This allows the domains in the 2nd generation to move ahead one more timestep before waiting again. After the children in the 2nd generation have moved N2 timesteps (the number of their timesteps per timestep of the uppermost parent) they can send upscale data to their parent which then allows the upper parent to take another timestep. Recall that while static nests can telescope without limit, a moving nest in the 2nd generation can have a single child within it but the 3rd generation nest cannot have children. The capability to have more than two generations of moving nests would add even more complexity to the code and was not done.

(5) A parent sends BC data to its children at the end of each parent timestep. For static nests the parents compute only once the association between their tasks and their children's boundary tasks then send those child tasks the information they need in order to be able to properly receive forecast data. Then the parents send the new boundary data to the child boundary tasks. For moving nests the parents are sent the new location of any of their children who moved. The parents then recompute the association between their tasks and their children's boundary tasks as well as with child tasks in the new region of the parent into which the children moved. Parents send the pertinent child tasks information they need in order to receive BC data. Finally the parents send their moving children new boundary data plus new internal data for the area of the parent newly covered by the most recent motion of the nests.

(6) Write out history/restart files if it is time to do so after the model's clock has been advanced at the end of the timestep.

After these processes complete then each task checks if it has finished the forecast in its current generation. When the task has completed the forecast in all the generations it is in then it exits the loop that continually iterates over all generations and it finalizes its execution.

6. Operational applications

The NMMB's nesting has been utilized operationally in two different ways. The first uses the static nests in the North American Model (NAM) forecast slot. A parent domain with 12 km grid spacing spans a large region over North America and extends west of Hawaii. Four nests with 3 km grid spacing lie on the parent. The four nests cover the CONUS, Alaska, Puerto Rico and Hawaii. In addition another static nest with 1 km grid spacing lies either on the CONUS or the Alaska nest. This nest can be relocated each cycle and is referred to as the Fire Weather nest since its original purpose was to provide the U.S. Forest Service with very high resolution

guidance for specific wildfire events. However it has also been placed over other areas of particular interest for such events as presidential inaugurations and the Super Bowl. All these nests are 1-way interactive with their parents.

The second application of the NMMB's nesting is in the operational Hurricanes in a Multi-scale Ocean-coupled Non-hydrostatic (HMON) model which is the atmospheric component of the coupled system that includes the Hybrid Coordinate Ocean Model (HYCOM). As described earlier the NMMB can have multiple moving nests that each contain its own moving nest. However the coupler employed in the HMON-HYCOM system has two stringent requirements. The first is that only a single moving outer nest with a moving inner nest be used on the upper parent domain thus the NMMB's capability to forecast multiple storms at once cannot be taken advantage of. The second is that the task layouts of the parent, outer nest, and inner nest must be identical thereby precluding the fine-tuning of task assignments on each domain in order to minimize clocktime. Given those requirements then each nest is 2-way interactive with its parent. Ocean coupling is explicit for the parent and outer nest domains. The inner nest's ocean-related surface variables are interpolated from the outer nest.

7. Summary

A comprehensive nesting capability was added to the NMMB. It allows the model domain to contain any number of static and/or moving nests the user wishes to use. Static nests can telescope to as many inner nests as the user desires. Moving nests can telescope to one deeper moving nest. All parent and child domains can employ either 1-way or 2-way interaction. The type of interaction chosen dictates the nature of the MPI task usage in order to make optimal use of resources and to minimize runtime. All tasks are uniquely assigned to individual domains when 1-way interaction is used so that when work is balanced properly all compute tasks are busy nearly all the time as all domains are proceeding in their forecasts concurrently. For 2-way interaction parent domains must always stop at the end of every timestep and wait to receive upscale data from their children. To compensate for these waits all compute tasks are uniquely assigned to the individual domains in what is considered to be the most computationally expensive generation of domains. Then in each remaining generation as many tasks as possible are reassigned across the domains of each of those generations avoiding the use of too many tasks on a domain which would slow down the run with too small decomposition subdomains. A given task can lie on domains in any or all generations but can only lie on one domain per generation. Then the generations execute sequentially while all domains within each generation execute concurrently. This means as many tasks as possible are busy at any given time allowing the forecast to run as fast as possible.

ACKNOWLEDGEMENTS

The author wants to thank Jim Abeles for the many interesting and useful conversations we had during the construction of the nests. Also thanks are extended to Jim and to Dusan Jovic for their thoughtful reviews of this manuscript.